University of Newcastle upon Tyne

COMPUTING LABORATORY

Very Large Distributed System Management (A Dependable Approach)

To appear, "International Workshop on Configurable Distributed Systems" Imperial College, London, 25-27 March 1992.

D.R. Hodge

TECHNICAL REPORT SERIES
No 375 February, 1992

TECHNICAL REPORT SERIES
No. 375 February, 1992

Very Large Distributed System Management (A Dependable Approach)

D.R. Hodge

Abstract

This paper details the dependability considerations applied to the development of a prototype system management workbench composed of a set of managed objects whose state corresponds to external managed resources. A method of incorporating manual control operations and coping with unconnected resources is presented.

Printed and published by the University of Newcastle upon Tyne,
Computing Laboratory, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

HODGE, Darren Richard

Very Large Distributed System Management (A Dependable Approach)
[By] D.R. Hodge

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Laboratory, 1992.

(University of Newcastle upon Tyne, Computing Laboratory,
Technical Report Series, no. 375)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.
Computing Laboratory. Technical Report Series. 375

Abstract

This paper details the dependability considerations applied to the development of a prototype system management workbench composed of a set of managed objects whose state corresponds to external managed resources. A method of incorporating manual control operations and coping with unconnected resources is presented.

About the author

D.R. Hodge joined the Computing Laboratory in 1988 as a CSSD student and is currently a demonstrator.

Suggested keywords

FAULT-TOLERANCE
MANAGEMENT
UNIX

Suggested classmarks (primary classmark underlined)

Dewey (18th):	001.64404	001.6425
U.D.C.	519.687	681.322.06

Very Large Distributed System Management.
(A Dependable Approach)

Darren R. Hodge
(D.R.Hodge@newcastle.ac.uk)

Computing Laboratory,
Department of Computing Science,
The University,
Newcastle upon Tyne
January 1992

ABSTRACT

This paper details the dependability considerations applied to the development of a prototype system management workbench composed of a set of managed objects whose state corresponds to external managed resources. A method of incorporating manual control operations and coping with unconnected resources is presented.

1. Introduction

Despite the fundamental role of distributed computer system management in the smooth running of any computer dependent organisation, it only recently that the nature and function of effective management techniques is being researched as a topic in its own right. Furthermore, current research is concentrated in Network Management [4,14], Organisational Modelling via domains [6,17] while the application of fault tolerance to prototype systems is often overlooked.

This paper is concerned with the dependability considerations associated with the automation of management activities in the context of very large distributed computer systems and in particular, those consisting of a wide variety of resources and vendors equipment and having an increasingly complex communications topology. That is where portability and standardisation are paramount in order to ensure compatibility and workability.

The paper is structured as follows :— Firstly an overview of structuring and fault tolerance techniques is presented. The prototype and working environment is presented with particular emphasis on applying control and monitoring operations to a line printing resource followed by critical evaluation and future work.

2. System Structuring

The model adopted for structuring the management information system combines current notations for describing organisations and enterprise modelling with the practicalities of fault tolerance. Enterprise modelling techniques have been drawn from the Ansa initiative [19] and the Esprit funded ORDIT project [13], and model the organisation in terms of :—

- *Agents* — representing a human or mechanical actor in an organisation,
- *Roles* — structural and functional. The distinction between structural and functional roles is largely concerned with the semantics and syntax of the functions played by agents. An example often used to illustrate this concerns legal and lexicographic proof reading whose tasks are similar but whose purposes are inherently different. [3]
- *Messages (Operations)* — used to convey control and status requests between functional roles and resources.
- *Resources* — comprising of a set of managed objects whose state reflects that of a managed resource. Resources are examined by agents and perceived according to their view. [11]

2.1. Fault Tolerant Duality

Entire systems can be structured equally well using both processes and message conversations (Process Model) and Objects and Atomic Actions (Object Model) and indeed form a fault tolerant duality.[10] However, traditionally, data base systems *tend* to be composed of atomic actions whereas process control applications *tend* to be composed of processes communicating with messages. In order to provide equivalence, the processes and message conversations must incorporate error recovery and checkpointing.

In the prototype system, this fault tolerant duality has been exploited and uses atomic actions as a structuring device for convenience. Both state and operation based recovery techniques can be employed to ensure dependability of managed objects together with compensation actions to undo the effects of operations upon external resources.[5]

(i) Example using Atomic Actions

An agent in her role as fault manager identifies herself to the management system. The fault manager is directly accountable to the computer services director (supervisor - subordinate) and has a team of operators responsible for the day to day running of computational resources. If the particular printer is found to be faulty (unable to boot automatically) it is reported to the manager, who in turn sends an operator to investigate. (i.e. the operator who has contractual obligations to manage that printer.) This scenario is encoded in the example below.

```
A.BEGIN_ACTION();
    Management System.Identify(agent);
    B.BEGIN_ACTION();
        if (Printer1.Boot() == ERROR)
        {
            Printer1.Compensate(FailedBoot);
            Agent1.Report(FailedBoot, Printer1);
            ...
            B.ABORT_ACTION();
        }
    B.END_ACTION();
A.END_ACTION();
```

In the pseudo code above (based on the C++ programming language[9]) it is assumed that objects A and B have been defined as Atomic actions, and have methods to BEGIN_ACTION, END_ACTION and ABORT_ACTION; Printer1 represents an

instance of a Printer object; Management System, is self explanatory. Within atomic action B, the agent attempts to Boot the printer, using the recoverable printer method; in the event of an error result being returned from the method, a compensation action is performed, the fault reported and the action is aborted. Atomic actions can be further sub-divided in terms of coloured, top-level and glued actions, and the interested reader is referred to Wheater's thesis.[18]

Several mechanisms are available to provide compensation, these include: (i) a set of undo operations together with state based recovery; (ii) operation based recovery via persistent operation log [5] and (iii) delaying external resource activity until the operation has committed its state.[7] Method (i) was selected for use in the prototype system as the use of operation based recovery has been evaluated in Dixon's thesis.

Example (ii) Processes and Messages

An agent in her role as change manager identifies herself to the management information system and is allowed to perform management activities associated with her contract. In order to disable a particular resource, menu options and icons are selected on her workbench. These functional roles are converted into message conversations and sent to the external resource controller via a secure communications channel.

Management control messages are received by the object store which in turn locks the managed object (or group of objects in the case of replica). Upon the completion of the update to the internal state of the object, the object store prepares to commit and the external resource is physically disabled. In the event of the external operation failing, compensation is performed and the conversation is aborted.

This scenario is outlined below :—

```
/*
        Object Store (server)
*/
A.Receive(Message, Workstation);
if (Message.Kind == StartRequest)
{
        A.BEGIN_CONVERSATION(ObjectStore, Workstation);
                A.Receive(TheRequest, TheResourse);
        A.PREPARE(CastYesVote);
                GetObject(TheResource);
                B.BEGIN_CONVERSATION(ObjectStore, TheResource);
                        B.Send(TheRequest, TheResource);
                B.PREPARE(CastYesVote);
                        UpdateState(TheRequest, TheResource);
                        B.Receive(TheVote, TheResource);
                        if (TheVote == CommitRequest)
                        {
                                B.COMMIT(); A.COMMIT();
                        }
                        else
                        {
                                // provide undo operation
                                ...
                                B.ABORT(); A.ABORT();
                        }
        }
```

```
/*
    Management workstation (client)
*/

A.BEGIN_CONVERSATION(Workstation, ObjectStore);
        A.Send(DisableRequest, Printer1);
A.PREPARE(CastYesVote)
        A.Receive(TheMessage, ObjectStore);
        if (TheMessage == CommitRequest)
                A.COMMIT();
        else
                A.ABORT();
```

In the above example, it is assumed that the BEGIN (PREPARE and COMMIT) and send (receive) operations are automatically logged, and the conversation is based upon a two phase commit protocol. [2] The management workstation in this example acts as a client of the object store, whose purpose is the hold the states of managed object. The session is commenced by the workstation BEGIN'ing a conversation with the object store and a *start request* is sent to the server. In turn, the server prepares to commit, updates the internal state of the object and echos the state transformation on the appropriate external object within a nested conversation. In the event of the operation being successfully completed, both actions commit otherwise a compensation action is performed to undo the effects of the external action and the nested action is aborted.

2.2. Object Location

Resources are located at each node in the distributed system and communicate via a remote procedure call mechanism to a management information centre located on a computer. Non computational resources are remotely controlled by probes housed on computational resources either connected to the network or using dedicated cable. A central object store is used to store managed object instance data. Although replication techniques can be employed, for the sake of simplicity, these have been omitted in the prototype system.

2.3. Control and Monitoring Operations

Control and monitoring operations to managed objects fall into three groups: those entirely automated, intermediate and entirely manual. For example, while it is possible to boot a resource automatically, *some* configuration operations such as installing a device require some manual intervention, while others, such as powering up a resource *must* be performed manually. Hence, in the development of the prototype system particular attention has been given to the practicalities of communicating with unconnected resources and performing manual tasks.

Processes and Messages Conversations form a fault tolerant duality of Objects and Actions, an indeed perform an abstraction of similar concrete operations. For example, BeginConversation ≡ BeginAction; MonitoringMessage ≡ ReadLock; ControlMessage ≡ WriteLock. (The interested reader is referred to Mancini's thesis) Hence it is possible to form an abstract conversation class from which atomic actions and message conversations are derived. It is further possible to derive a manual conversation class in order to contact a (human) agent, using for example, electronic mail as a communications media.

3. Development of the Prototype

The prototype system is written in the C++ Programming Language where each managed object corresponds to an instance of a C++ class, and uses the fault tolerance capabilities of the Arjuna Distributed Programming Tool Kit.[16] This (multiply) hierarchy of objects is described in the sections below:

3.1. Prototype Components

Components of the management system include a set of managed objects and external resources 'typical' of those found in any computer intensive organisation. These include:—

- *managed object* — which acts as a base class from which other managed resources are derived. A method is provided to report failures, which can be redefined by derived objects. Instance variables include an object identifier, location and nature of object (i.e. internal or external)
- *device* — representing peripheral devices linked (or attached to) computational resources or the communications subsystem. Devices can be powered up (down) as well as booted (halted). Instance variables include port details such as port configurations, line speeds and security attributes. These have been abstracted from the Encore annex network administrators guide 20 as a common subset of datum required to configure terminals, printers and indeed modems.
- *printer* — representing printer peripheral devices. Printer devices are inherited from the device class. Methods include the ability to boot (shutdown) the resource, restart and obtain the status of the printer. These operations have been abstracted from the */etc/lpc* command available on the UNIX† operating system. Instance variables include: paper attributes (size, volume printed etc.), spooler configuration (reachability, location, spool queue). Particular idiosyncrasies such as printcaps, downloading fonts and graphical definition systems have been deliberately omitted from the model.
- *terminal* — representing glass teletype (tty) devices and derived from the device class. Methods include the ability to set (get) tty attributes
- *agent* — a human (or mechanical resource) which has the ability to perform management control (and monitoring) operations.

For each of the managed objects associated with external resources, further objects are defined to provide external compensation actions. For example, the class RecoverableDevice is derived from Device and use the Device methods as an abstraction of the external (non recoverable) device to which dependability is added.

3.2. Hardware Configuration

A Sun workstation (3/60) running the 4.1.1 release of the *SunOs* operating System is employed as a systems management workbench and is connected to the University campus Ethernet.[12] Hardware peripherals (glass teletype terminal and line printer) are configured as pseudo-devices within the sun workstation and are in fact connected to an Encore Annex network terminal concentrator. [21] Although it is possible to physically connect these resources directly to the workstation using serial port connections, it was far more practical to locate the devices in a laboratory machine room under

† UNIX is a registered trademark of Unix System Laboratories.

controlled temperature and reliable power supply.

4. Prototype Architecture

In order to discuss the architecture of the management information system, let us adopt a bottom up approach: firstly examining the integration of managed objects and external resources; then agents in roles accessing those resources in a regulated manner.

4.1. Integrating External Resources

This forms the base layer of the management information system and allows the integration of managed objects with their external resources. Let us consider the device class, as defined below:

```
class Device
{
public:
        ...
        virtual Error   PowerUp();
        virtual Error   PowerDown();

        virtual Error   Connect();
        virtual Error   Disconnect();
        virtual Error   Boot();
        virtual Error   Reset();
        virtual Error   GetStatus();

protected:
        DeviceName          TheDeviceName;
        DeviceState         TheState;
        PortDetails         ThePortDetails;
        NetworkDetails      TheNetwork;
        LineSetting         TheLine;
        DeviceMode          TheMode;
        DeviceKind          TheDeviceKind;
        SecurityDetails     TheSecurity;
};

class RecoverableDevice: public Device, public ManagedObject
{
public:

        RecoverableDevice(Uid * Unew, Error &);
        RecoverableDevice(Uid * Uold, Error &);
        ...

        virtual save_state(ObjectState *, object_type);
        virtual restore_state(ObjectState *, object_type);

        virtual TypeName    type();
};
```

Objects in Arjuna are referenced using unique identifiers (Uid's) and their state is made persistent using save (restore) methods. A name server is used to map the abstract typename and Uid to the physical object located at a particular node. Stubs are generated automatically and methods are invoked using a remote procedure call. Hence, the *RecoverableDevice* class inherits the methods of both the *Device* and *ManagedObject* classes and supplies methods to both save and restore the objects state.

The two object constructors allow the creation of a new object and retrieval of a persistent object (respectively).

It is clear from the types of methods which the class *Device* provides that methods fall into several categories; these include: fully manual, such as powering up (down); intermediate, such as connecting (disconnecting); and fully automatic, such as getting the device status. Communication between managed objects and their external counterpart have been described in terms of a class *Operation* from which are derived *ManualOperation* and *AutomaticOperation*. Intermediate operations, such as device connection can be composed using both manual and automatic operations. Intermediate operations are composed of both manual and automatic components, and hence use both *ManualOperation* and *AutomaticOperation* methods.

```
class Operation
{
public:
        Operation(OperationName, LogFile);
        ~Operation();

        virtual ReadFrom(PeramList, Results &);
        virtual WriteTo(PeramList);
        virtual ReadWrite(PeramList, Results &);
protected:
        Operation       TheOperation;
        LogFile         TheLogFileName;
};
```

ManualOperation and *AutomaticOperation* classes have been derived from the *OperationClass* and have redefined methods for communicating with external resources; this is explained in detail later. The *Operation* class provides facilities for operation logging and records a textual transcript of operations performed upon external resources by a particular agent.

Where possible, existing system software has been integrated within the prototype to enact fully automatic operations, these include the Line Printer Controller (*lpc*), Encore Network Administration *na*, and in the case of fully manual operations, an electronic mail protocol is used to contact the agent responsible for the resource.

4.1.1. Fully Automatic Operations

System software has been integrated using system calls available on the UNIX operating system. Thus in the case of batch commands, the command line is constructed by concatenating the absolute command name to its parameter list and issuing a pipe command to the operating system shell, and reading its results or writing parameters. This is illustrated in the program fragment below. Note that in order to perform some line printer configuration commands UNIX *super user* privileges are required.

```
AutoCommand::ReadFrom(ParamList TheParams, Results & TheResults)
{
        ...
        FILE *CommandStream;

        strcpy(TempCommand, TheCommand);
        strcat(TempCommand, TheParams);
        ...
        CommandStream = popen(TempCommand, "R");
        ...
        fscanf(CommandStream, ScanSpec, TheResults);
        ... pclose(CommandStream);
}
```

Unfortunately, the *popen* command cannot be used in the case of interactive commands. Interactive commands (such as *na*) are integrated essentially by executing the command as a background process and reading (writing) parameters, as illustrated in the program fragment below.

```
AutoCommand::ReadWrite(ParamList TheInputs, Results & TheOutputs)
{
        int     ParentToChild[2];
        int     ChildToParent[2];
        int     TheChild;

        SystemCall(pipe(ParentToChild));
        SystemCall(pipe(ChildToParent));
        SystemCall(TheChild = fork());

        if (TheChild == 0)
        {
                /* child code */
                ...
                SystemCall(dup2(ChildToParent[1], 1));
                SystemCall(dup2(ChildToParent[1], 2));
                SystemCall(dup2(ParentToChild[0], 0));

                execl(ThePath, TheCommand, (char *) 0);
        }
        else
        {
                /* parent code */

                WriteBuffer(ParentToChild[1], TheInputs);
                ReadBuffer(ChildToParent[0], TheOutputs);
        }
}
```

This function uses the *fork* system call to divide the process into a child and parent. The child process redefines the pipes to read (write) to access the standard input (output and error) streams of the executed process. (i.e streams 0, 1 and 2 of the process table.) Meanwhile the parent process writes (reads) input (output) parameters to (from) the process. The interested reader is referred to [8, 15]

4.1.2. Fully Manual Operations

In order to perform fully manual operations the (human) agent responsible for managing the particular resource is contacted using an interface to the electronic mail system.

Hence, in order to boot a particular device, the management workstation determines which agent is responsible for managing the resource, mails her, and requests that the device be booted. The actual protocol for describing successful (or otherwise) completion of operations in terms of textual messages is somewhat primitive, but attempts to simulate the outcome of automatic operations performed using system commands.

4.2. Agent to Managed Objects

The interface between agents and managed objects is somewhat primitive, and described in terms of a program's interface to the management information system, as outlined in the program fragment below:—

```
A.BEGIN_ACTION();
        ...
        if (TheManagementSystem.Identify(Agent1) == OkAgent)
        {
                Agent1.Adopt(ConfigManager);
                ...
                Printer1.Boot();
        }
A.END_ACTION();
```

Initially, the agent *Agent1* identifies herself to the management information system and is allowed to adopt several roles depending upon her contract. A contract database is being developed, and consists of a total mapping between agent—role tuples and is used for contacting agents in the event of failures and requesting manual intervention. Thus *agent1* adopts the role of configuration manager and boots *printer1*.

5. Case Study: Device Installation

Device installation requires the following operations :— Firstly, the device is physically connected to an Annex terminal concentrator port (manually), and appropriate port settings are configured. These parameters typically include the line speed, control and parity bits and line mode (either simplex or duplex). Secondly, a pseudo device is created on the host resource, which allocates a device identifier (typically of the form /dev/tty**) and a port number determined. An entry is then placed in the appropriate configuration file (/etc/termcap or /etc/printcap). Finally, a remote telnet connection is established linking the pseudo device to the physical resource.

These operations are performed at several layers of the management system. Note that in the example code described below, it is assumed that objects have been defined describing Device, Printer, RecoverableDevice and RecoverablePrinter. The atomic action methods have been simplified in order to mask the Arjuna specific details of starting actions, acquiring read (write) locks etc. The interested reader is referred to the Arjuna programmers guide for further information.[1]

```
PrinterError    RecoverablePrinter::InstallPrinter()
{
        ...
        A.BEGIN_ACTION();
                TheDeviceError = InstallDevice();
                ThePrinterError = Printer::InstallPrinter();
                ...
                if (ThePrinterError != OkPrinter)
                {
                        Compensate(BadInstall);
                        A.ABORT_ACTION();
                        ...
                }
        A.END_ACTION();
}
```

In the event of nodes failing within the atomic action, compensation must be performed upon node recovery. This will be performed using an object constructor which reads the state of the persistent object and compares this state with the actual state of the external object. Other methods of achieving this effect include the construction of a persistent operation log, as illustrated in Dixon's thesis.

6. Conclusions and Future Work

This paper has discussed the dependability considerations required when applying fault tolerance techniques to the area of distributed system management. Typically such process control systems have tended to be structured using the process — message conversation model and this paper has illustrated the use of objects and actions to construct state based recovery. Compensation operations have then been added in order to provide error recovery to external resources.

With regard to compensation actions, it can be argued that the construction of a persistent operation log and full operation based recovery presents a superior method of achieving dependability of external resources. Both methods require a set of *anti* or *undo* operations to provide error recovery, and persistence of instance datum; thus both achieve the same effect and complement each other.

Having developed a prototype management information system, several avenues of research are proposed. These include the investigation of a mechanism for recording and planning configuration changes *Expected Exceptions* and further expanding the scope of the prototype system.

7. Acknowledgements

This work was part funded by SERC award reference: 8830487X. The author gratefully acknowledges the technical assistance of colleagues at The University of Newcastle upon Tyne in supplying and installing the hardware components of the prototype system, in particular to Tim Smith and Steve Varty, Adrain Waterworth for proof reading drafts of this document, and Dr. Lindsay Marshall for supervising this project.

References

1. *The Arjuna Reference Manual (Version 2),* The University of Newcastle upon Tyne, 1992 .

2. Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman, in *Concurrency Control and Recovery in Database Systems*, p. Addison Wesley, 1987.
3. Andrew Blyth, *Role Relation Diagrams,* University of Newcastle upon Tyne, Newcastle upon Tyne, England, 1989. MSc Thesis
4. J. Case, M. Fedor, M. Schoffstall, and J. Davin, ''A Simple Network Management Protocol,'' *RFC*, vol. 1067, Augest 1988.
5. G.N. Dixon, *Object Management for Persistence and Recoverability,* University of Newcastle upon Tyne, Newcastle, UK, December 1988. Phd Thesis
6. ISO, ''OSI-Basic Reference Model: Part 4 Management Framework,'' *ISO/IEC/DIS 7498-4*, March 1988.
7. P.A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice,* Springer-Verlag, New York, 1990.
8. Samuel J. Leffler, ''An Advanced 4.3 BSD IPC Tutorial,'' *PS 1: 8*, UNIX.
9. Stanley B. Lippman, *C++ Primer,* Addison Wesley, 1989.
10. L.V. Mancini, *Reliability Issues in the Design of Distributed Object-Based Architectures,* University of Newcastle upon Tyne, Newcastle, UK, September 1989. Phd Thesis
11. Lindsay F. Marshall, ''Managing Management - The TOBIAS Approach,'' in *ESPRIT 1990 Conference*, 1990.
12. Robert A. Metcalf and David R. Boggs, ''Ethernet: Distributed Packet Switching for Local Computer Networks,'' *CACM*, vol. 19, no. 7, pp. 395 - 404, ACM, July 1976.
13. ORDIT, *Draft Technical annex* , ESPRIT, 1989.
14. Craig Partridge and Glenn Trewitt, ''The High Level Entity Management System (HEMS),'' *IEEE Network*, vol. 2, no. 2, pp. 37 - 42, IEEE, March 1988.
15. Stuart Sechrest, ''An Introductory 4.3 BSD IPC Turorial,'' *PS 1: 7*, UNIX.
16. S.K. Shrivastava, G.N. Dison, and G.D. Parrington, ''An Overview of Arjuna - A Programming System for Reliable Distributed Computing,'' *IEEE Software*, IEEE, (to appear).
17. Morris Sloman and Jonathan D. Moffett, *Domain Management for Distributed Systems*, 22nd August 1988. to be presented at IFIP Integrated Network Management Sysp, Boston,
18. Stuart M. Wheater, ''Constructing Reliable Distributed Applications using Actions and Objects,'' *Ph.D. Thesis*, Newcastle University, September 1989.
19. J. Winterbotham, ''Management,'' in *ANSA Reference Manuel*, vol. iii, ANSA, June 1987.
20. ____, ''Annex Network Administrators Guide,'' 716-02885, Encore, 1987.
21. ____, *Annex ii Hardware Installation Guide,* Encore, 1987.